

NO-A167 069

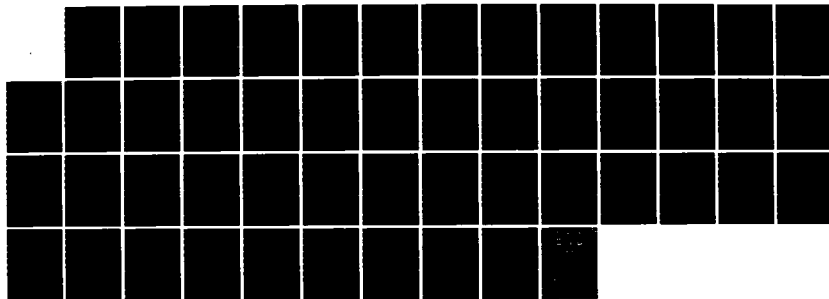
HARDWARE IMPLEMENTATION OF A CONCATENATED
ENCODER/DECODER(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING
P W GRAAF DEC 85 AFIT/GE/ENG/85D-12

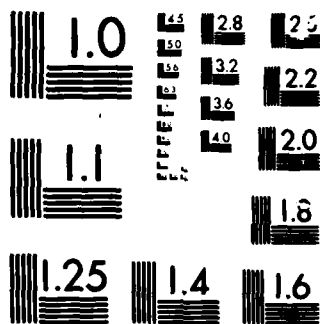
1/1

UNCLASSIFIED

F/G 9/5

NL





MICROCOPY

CHART

AD-A167 069



HARDWARE IMPLEMENTATION OF A
CONCATENATED ENCODER/DECODER

THESIS

Peter W. de Graaf
Captain, USAF

AFIT/GE/ENG/85D-12

DISTRIBUTION STATEMENT A

Approved for public release
Distribution Unlimited

DTIC
ELECTE
MAY 12 1988

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

86 5 12 028

DTIC FILE COPY

AFIT/GE/ENG/85 *D-12*

HARDWARE IMPLEMENTATION OF A
CONCATENATED ENCODER/DECODER

THESIS

Peter W. de Graaf
Captain, USAF

AFIT/GE/ENG/85D-12

DTIC
ELECTE
MAY 12 1986
S B D

Approved for public release; distribution unlimited

HARDWARE IMPLEMENTATION OF A CONCATENATED ENCODER/DECODER

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering

Peter W. de Graaf, B.S.

Captain, USAF

December 1985

Approved for public release; distribution unlimited

Preface

The ultimate goal of this study and the follow-on studies which will use the circuits I have designed is to calculate the error detection and correction performance of different concatenated coding schemes. Bounds for the error rates of these different coding schemes can be calculated; however, these bounds are not tight enough to be useful. The actual error rates may be two or three orders of magnitude better than the calculated bound. Thus, given an adequate model for the errors on a channel and a specific concatenated coding scheme, the actual error rate for a given situation can be determined.

While working on this project, I have received a great deal of help and support from others. First, I thank my Lord Jesus Christ for sustaining me. "Unless the Lord builds the house, they labor in vain who build it" (Psalm 127:1). I also thank my advisor, Captain Glenn Prescott, for his patience with me and for his valuable assistance. And I thank Major Ken Castor for his insights to coding theory and for helping me get started on this project. Special thanks to Mr. Orville Wright for helping me with the circuit boards. A word of thanks to Mr. Frisky for his counsel and support. Finally, a special thanks to my loving wife Betsy for her understanding and her prayers, and for typing and editing my thesis. "An excellent wife, who can find? For her worth is far above jewels. The heart of her husband trusts in her, and he will have no lack of gain" (Proverbs 31:10,11).

Peter W. de Graaf

Table of Contents

	Page
Preface	ii
List of Figures	iv
Abstract	v
I. Introduction	1
II. Code Selection	5
Criteria	5
Binary BCH Codes	5
Golay Code	6
Reed-Solomon Codes	7
Binary Convolutional Codes	7
Interleaving/Deinterleaving	8
III. Circuit Descriptions	9
BCH and Golay Encoder	9
BCH and Golay Decoder	13
Interleaver/Deinterleaver	21
Convolutional Encoder	25
IV. Operations	30
Data Transfer	30
General Configuration	31
V. Conclusions and Recommendations	32
Appendix A: BCH/Golay Encoder Assembly Language Routine	33
Appendix B: BCH/Golay Decoder Assembly Language Routine	34
Appendix C: Interleaver/Deinterleaver Assembly Language Routine	37
Appendix D: Convolutional Encoder Assembly Language Routine	38
Bibliography	39
Vita	40

List of Figures

Figure	Page
1. BCH/Golay encoder controller circuit	10
2. BCH/Golay encoder circuit	11
3. BCH/Golay decoder controller circuit	15
4. BCH/Golay jumper registers	16
5. BCH/Golay syndrome register	17
6. Interleaver/deinterleaver circuit	23
7. Convolutional encoder controller circuit	26
8. Convolutional encoder circuit	27

Accession	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	<input type="checkbox"/>
By	
Date	
Availability	
Notes	
A-1	

Abstract

This study describes the hardware implementation of a concatenated error correcting encoder/decoder. Individual burst and random error correcting coders were implemented using standard TTL integrated circuits and Z-80 microprocessors. The circuits handle input and output operations with a three line handshake. Thus, data transfer between circuits is asynchronous, and the coders may be concatenated in any order.

Reed-Solomon, BCH, Golay, interleaving, and convolutional codes were considered. Of these codes, the BCH encoder/decoder, the Golay encoder/decoder, the interleaver/deinterleaver, and the convolutional encoder were all implemented in hardware. The Reed-Solomon encoder/decoder and the convolutional decoder will be implemented in a follow-on study in software.

This study is the first part of a group of studies which will ultimately determine the actual error detection and correction performance of various concatenated coding schemes.

HARDWARE IMPLEMENTATION OF A CONCATENATED ENCODER/DECODER

I. Introduction

Background

Digital communications have significantly improved our ability to transmit large volumes of data in a short amount of time. However, digital communication systems are susceptible to noise. Specifically, noise in the communication channel causes errors by changing the values of the transmitted digits. For example, assume that a transmitted sequence is 01000011. This is the binary ASCII code for a "C". If an error occurs in the first bit position, the received sequence is 01000010. This is the binary ASCII code for a "B". Without additional bits for error detection, the receiver has no way to know that an error has occurred, and it accepts a "B" when it should have accepted a "C".

Several codes have been developed to detect and correct errors which occur during transmission. There are two basic types: random error correcting codes and burst error correcting codes. Random errors are single digit errors which occur randomly in the transmitted digits, and burst errors are groups of errors which occur randomly in the transmitted digits. Each type of code works well for correcting one of these types of errors. However, no single code has been developed that works well for correcting both types of errors.

This report is the first part of a study which will determine the specific capabilities of concatenating burst and random error correcting

codes. The overall approach is to use hardware and software encoders to encode a sequence of bits, transmit the encoded bits over a software simulated channel, and decode the received bits using hardware and software decoders. The decoded sequence will then be compared to the original sequence, and the error correction capabilities of the coding will be calculated.

Problem and Scope

The purpose of this study is to design and build a hardware encoder/decoder that uses concatenated burst and random error correcting codes. The encoder will be set up as a burst error correcting encoder followed by a random error correcting encoder. The decoder will be set up as a random error correcting decoder followed by a burst error correcting decoder. Reed-Solomon, BCH, Golay, Viterbi, and interleaving codes will be considered in this study. The codes not implemented in hardware will be implemented in a follow-on study using software techniques. Evaluation of the encoder/decoder's performance will not be accomplished in this study. A follow-on study will analyze the burst and random error correcting capabilities of the encoder/decoder.

Assumptions

The encoder/decoder will be implemented as a binary symbol channel encoder/decoder. The input to the encoder is assumed to be the binary sequence from the output of a source encoder. Thus, the encoder will not implement the source coding algorithm. This is a valid assumption since error correction is handled by the channel encoder and is not affected by the source coding algorithm.

Approach

Each code will be evaluated to see how easily it can be implemented using hardware techniques. Those codes which require complex encoding and decoding algorithms will be implemented in a follow-on study using software techniques. The hardware encoder/decoder will run at a higher data rate than the software encoder/decoder, allowing more data to be collected in a given amount of time. Thus, as many of the codes as possible will be implemented using hardware techniques.

The encoder/decoder circuits will be implemented using standard digital logic circuits and microprocessor controllers. Circuit development will be accomplished using protoboards. When circuit development is complete, the circuits will be implemented on a printed circuit boards.

The purpose of this report is to explain how these circuits work and how they can be connected to operate in a concatenated coding scheme. Also, a brief explanation of the theory for each code is presented. A list of references which explain the coding theories in detail is presented in the bibliography. Not all of these references are cited in this report, but they are included to provide a more complete list of reference material.

Chapter II contains a brief description of the theory behind each code and an explanation of which codes will be implemented in hardware. Chapter III is an explanation of how each circuit works. Chapter IV describes how the circuits work in a concatenated scheme. Chapter V contains conclusions and recommendations for follow-on projects.

Materials and Equipment

All of the components and equipment necessary to build the encoder/decoder are available in the communications laboratory.

II. Code Selection

Criteria

The selection of which codes will be implemented in hardware is based on two criteria. First, the coding and decoding algorithms must be compliant to a hardware implementation. Some algorithms require many conditional branches and decisions and are too complex to be implemented in hardware. Second, the implementation of the coding and decoding algorithms must be flexible enough to allow selection of different code parameters, such as input block length, output block length, character size, and generator coefficients. The coding and decoding algorithms which do not meet these criteria will be implemented in software in a follow-on study.

Binary BCH Codes

The BCH codes are a generalization of the Hamming codes for multiple error correction (1:141). As defined in [1], for any positive integers m ($m > 2$) and t ($t < 2^{m-1}$), a binary (n, k) BCH code has a block length n (equal to $(2^m - 1)$), $(n - k)$ parity-check bits, and a minimum distance d (greater than or equal to $(2t + 1)$). This code can correct any t or fewer errors in an n -bit block (1:142).

The binary BCH codes have a simple encoding algorithm (1:141). For an (n, k) code this algorithm can be implemented around a binary shift register of length $(n - k)$. The outputs of the shift register cells which correspond to the non-zero coefficients of the generator polynomial are summed (modulo two) to form the output. To change any of the code parameters requires only a change in the taps from the shift register

cells to the summer circuit. Thus, the encoding algorithm can be implemented in a circuit which allows selection of different code parameters.

There are many different decoding algorithms for binary BCH codes. Some of these algorithms, such as the Meggitt algorithm (1:104-6), will achieve the complete error detection and correction capabilities of the BCH codes. However, refinements are necessary for practical implementation of these algorithms (1:125). Error-trapping decoding is one such variation of the Meggitt algorithm. The error-trapping algorithm can be implemented with combinational logic circuits and a microprocessor controller. This algorithm is very effective for decoding single error correcting codes, short double error correcting codes, and burst error correcting codes (1:125). The error-trapping decoder can be built around a divider circuit with feedback taps set to correspond to the coefficients of the code's generator polynomial. Thus, similar to the encoding algorithm, the decoding algorithm can be implemented in a circuit which allows selection of different code parameters.

Golay Code

The Golay code is a cyclic binary code which is similar to the BCH codes. It is a (23,12) code and the only known multiple error correcting binary perfect code which can correct any combination of three or less errors in a block of 23 bits (1:134). The only difference between the Golay code and the BCH codes is the coefficients in the generator polynomials. Thus, both the encoding and decoding algorithms can be implemented in hardware exactly as the BCH codes.

However, it is important to note that there are two algorithms for

decoding Golay codes which will achieve the complete error detection and correction capabilities of these codes. These are the Kasami (1:135) and systematic search (1:138) algorithms. The Kasami algorithm requires multiple arithmetic and logical operations, and the systematic search algorithm requires complex clock and timing circuitry. Both of these algorithms would require extensive programming of the microprocessor controller in the decoder circuit. Thus, these are best implemented in software and may be implemented in a follow-on study.

Reed-Solomon Codes

Reed-Solomon codes are a type of non-binary (q -ary) BCH codes (1:170). As defined in [1], a t error correcting Reed-Solomon code has a block length of n characters (equal to $(q - 1)$), $(n - k)$ parity-check characters (equal to $2t$), and a minimum distance d (equal to $(2t + 1)$).

The encoding and decoding algorithms are similar to those for the binary BCH codes with one major difference: all of the arithmetic for the q -ary codes is done modulo q . Thus, all of the adding, multiplying, dividing, and shifting operations are done on a character-by-character basis instead of a bit-by-bit basis. This would require very complex and extensive circuitry to implement in hardware. Also, each code would require a separate circuit. Thus, the Reed-Solomon encoder and decoder are best implemented in software.

Binary Convolutional Codes

An (n, k, m) convolutional code differs from an (n, k) block code in that the n bits out of the encoder at any given time depend on both the k input bits at that time and the previous m input bits (1:287).

Basically, the information bit sequence is convolved with the $(m + 1)$ coefficients of the n generator sequences, using k bits per shift.

The binary convolutional codes have a simple encoding algorithm (1:287-95). For an $(n, 1, m)$ convolutional code this algorithm can be implemented around a binary shift register of length m . The outputs of the shift register cells which correspond to the non-zero coefficients of the code generators are summed (modulo two) to form the n outputs. To change any of the code parameters requires only a change in the taps from the shift register cells to the summer circuit. Thus, the encoding algorithm can be implemented in a circuit which allows selection of different code parameters.

The decoding algorithm for convolutional codes selects the bit sequence which was most likely the transmitted sequence (1:315-22). This decision involves many conditional branches and requires the decoder to keep track of multiple paths through a trellis diagram. Thus, the convolutional decoder can only be implemented in software.

Interleaving/Deinterleaving

Interleaving is a process used to spread the n output bits of any code over a sequence of bits much larger than n . Essentially, this spreads a burst error over a number of codewords instead of just one codeword.

Both the interleaver and deinterleaver involve simple algorithms (1:271-2). Both can be implemented around a storage register and a microprocessor controller. The assembly language program required to run the controller is very short. Thus, both the interleaver and deinterleaver will be implemented in hardware.

III. Circuit Descriptions

BCH and Golay Encoder

As stated in Chapter II, the BCH and Golay encoders can be implemented by the same circuit design. The only difference is in the coefficients of the generator polynomials. Figure 1 is a diagram of the controller circuit, and Figure 2 is a diagram of the encoder circuit.

The heart of the controller is the Z-80 microprocessor. The 2716 EPROM stores the assembly language routine which runs the Z-80. Since RAM is not used in this circuit, the MREQ* signal (the "*" indicates an active-low signal) is connected directly to the CE* of the 2716. Thus, the 2716 is enabled on any memory operation. The RD* signal is connected to the OE* and is used to gate the memory contents onto the data bus during a memory read operation. Address lines A0 through A10 are connected directly to the 2716. This provides 2048 memory locations (0000H to 07FFH) for the assembly language routine. Data lines D0 through D7 are connected directly to the 2716. The clock signal is provided by a crystal oscillator. The output of the oscillator is a 2.45 MHz square wave.

All of the control functions are implemented through input/output (I/O) ports. The IORQ* signal is connected to the G2A* enable of the '138 (three to eight line decoder). Thus the '138 is enabled on any input or output operation. Address lines A0 through A2 are connected to the three line input on the '138. This provides control for eight I/O ports (00H to 07H).

The following is a step-by-step description of how the controller works. When a reset signal is received, the Z-80 starts the program from

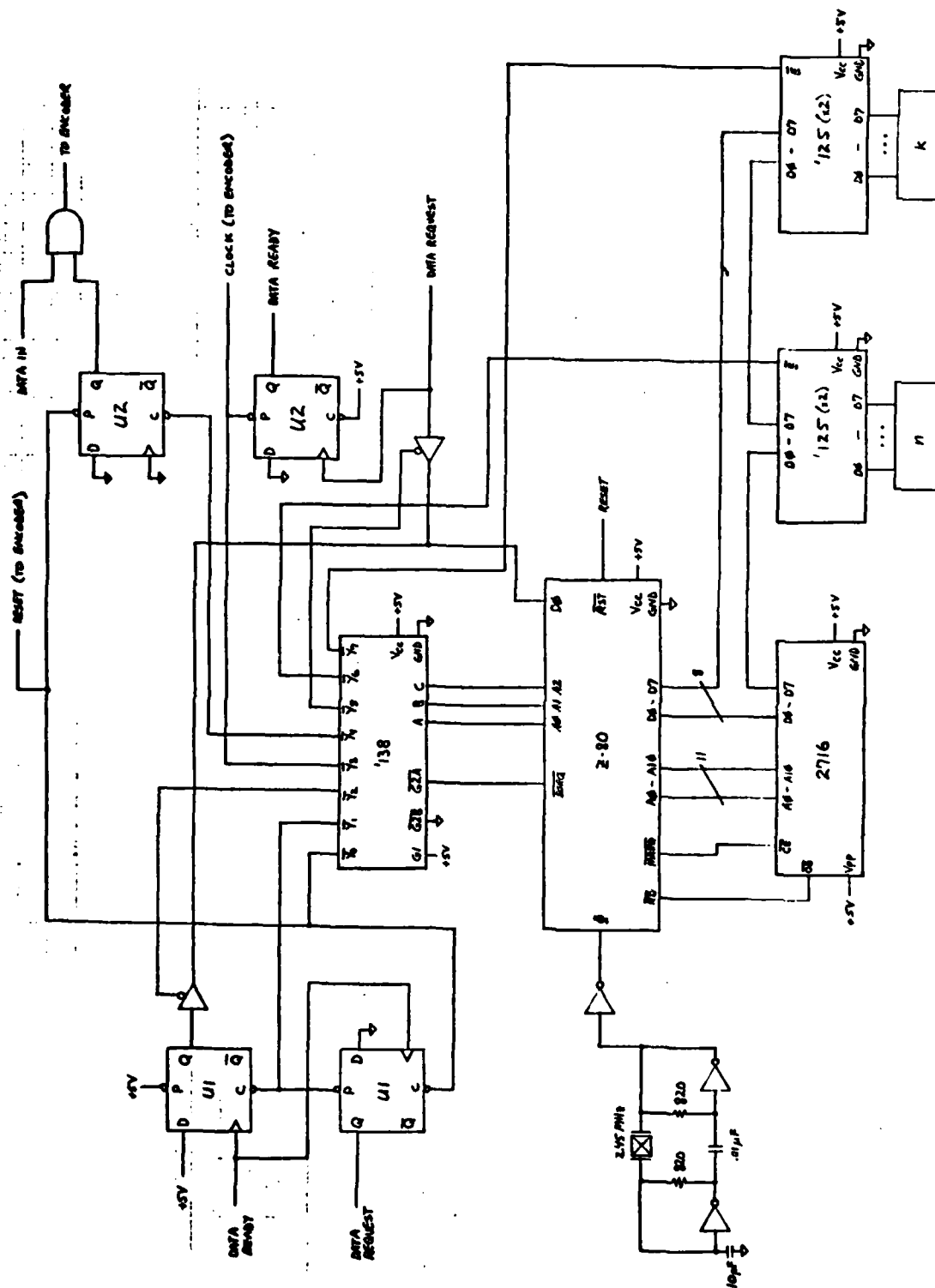


Figure 1. BCH/Golay encoder controller circuit

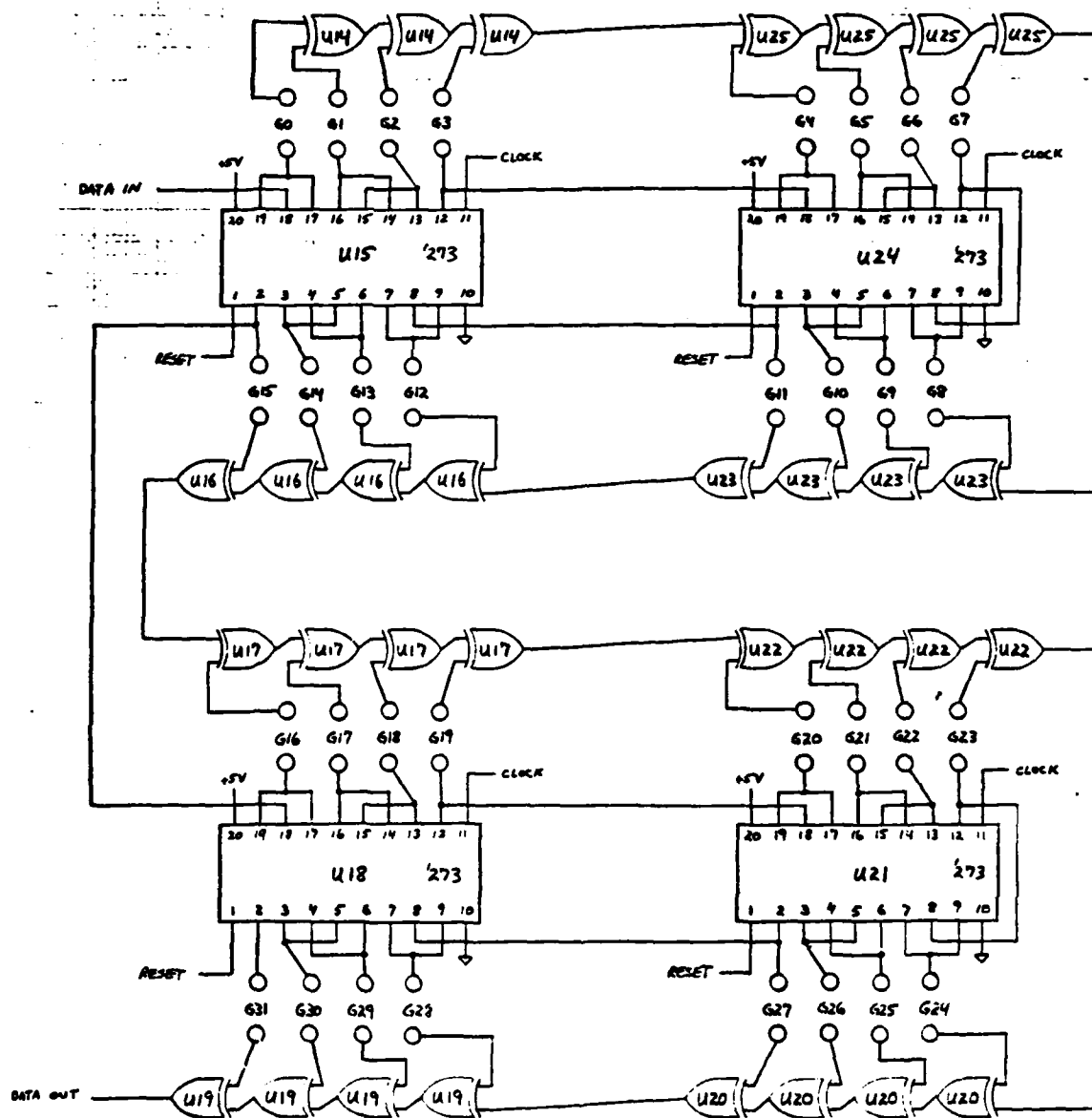


Figure 2. BCH/Golay encoder circuit

memory location 0000H. The assembly language routine which runs the Z-80 is shown in Appendix A.

Step 1. The controller strobes I/O port Y0*. This clears the input data request flip-flop U2 and clears the registers in the encoder circuit.

Step 2. The controller inputs the value for n from I/O port Y6*. This value is preset to the desired output block length (in binary) by jumpers before encoding operations begin.

Step 3. The controller inputs the value for k from I/O port Y7*. This value is preset to the desired input block length (in binary) by jumpers before encoding operations begin.

Step 4. Data bit D0 is input from I/O port Y5*, and its value is checked. This is the output data request signal. If D0 is zero, step 4 is repeated. If D0 is one, the program proceeds to step 5.

Step 5. The controller strobes I/O port Y1*. This clears the input data ready flip-flop U1 and sets the input data request flip-flop U1.

Step 6. Data bit D0 is input from I/O port Y2*, and its value is checked. This is the input data ready signal. If D0 is zero, step 6 is repeated. If D0 is one, the program proceeds to step 7.

Step 7. The controller strobes I/O port Y3*. This clocks the data bit into the encoder circuit and sets the output data ready flip-flop U2.

Step 8. The controller checks the total number of data bits clocked into the encoder circuit. If this is less than k, the program goes back to step 4. If this is equal to k, the program proceeds to step 9.

Step 9. The controller strobes I/O port Y4*. This disables the data gate flip-flop U2 so that the input to the encoder circuit is a zero.

Step 10. Data bit D0 is input from I/O port Y5*, and its value is checked. This is the output data request signal. If D0 is zero, step 10 is repeated. If D0 is one, the program proceeds to step 11.

Step 11. The controller strobes I/O port Y3*. This clocks the zero bit into the encoder circuit and sets the output data ready flip-flop U2.

Step 12. The controller checks the total number of data bits clocked into the encoder circuit. If this is less than n , the program goes back to step 10. If this is equal to n , the program goes back to step 1.

Thus, the basic functions of the controller are to handle the interfacing with the other circuits and to insure that k information bits followed by $(n - k)$ zeros are clocked through the encoder.

The encoder circuit is a series of D flip-flops (U15, U24, U18, U21) with their outputs jumpered to an exclusive OR summer circuit. These jumpers (30 through 331) are set to correspond to the binary coefficients of the desired generator polynomial. Each of the inputs to the exclusive OR gates has a pull-down resistor to provide a zero input when the input is not jumpered (i.e., the corresponding generator coefficient is zero). This prevents the input from floating when it is not connected to the output of a flip-flop. The RESET, CLOCK, and DATA IN signals all come from the controller circuit, as previously explained. The DATA OUT signal goes to the next encoder circuit.

BCH and Golay Decoder

Just as the BCH and Golay encoders are implemented by the same circuit, so are the BCH and Golay decoders. Again, the only difference is in the coefficients of the generator polynomials. Figure 3 is a

diagram of the controller circuit, Figure 4 is a diagram of the jumper registers, and Figure 5 is a diagram of the syndrome register.

The controller in the decoder circuit is set up similar to the controller in the encoder circuit. The only major difference is that the controller in the decoder circuit uses a RAM working memory in addition to the EPROM.

For the EPROM, address lines A0 through A10 are connected directly to the 2716. The output from ORing A14 and MREQ* is connected to the CE* of the 2716. RD* is connected to the OE*. This set-up provides 2048 memory locations (0000H to 07FFH) for the assembly language routine. Data lines D0 through D7 are connected directly to the 2716. For the RAM, address lines A0 through A9 are connected directly to both 2114's. The output from ORing A14* and MREQ* is connected to the CS* of the 2114's. This provides 1024 memory locations (4000H to 43FFH) for the Z-80 to use. The WR* signal is connected to the WE* signal of the 2114's. This controls the read/write operations in the RAM. Data lines D0 through D3 are connected to one 2114, and D4 through D7 are connected to the other 2114. Since each 2114 has only a 4-bit memory word, this provides a full 8-bit memory word for RAM operations. The clock signal for the Z-80 is provided by the crystal oscillator. The output of the oscillator is a 2.45 MHz square wave.

All of the control functions are implemented through the I/O ports. Address lines A0 through A3 are connected to both '154's (four to sixteen line decoder). A4 and IORQ* are used to enable the first '154, providing control signals 00* through 0F*. A4* and IORQ* are used to enable the second '154, providing control signals 10* through 1F*. Thus, control of

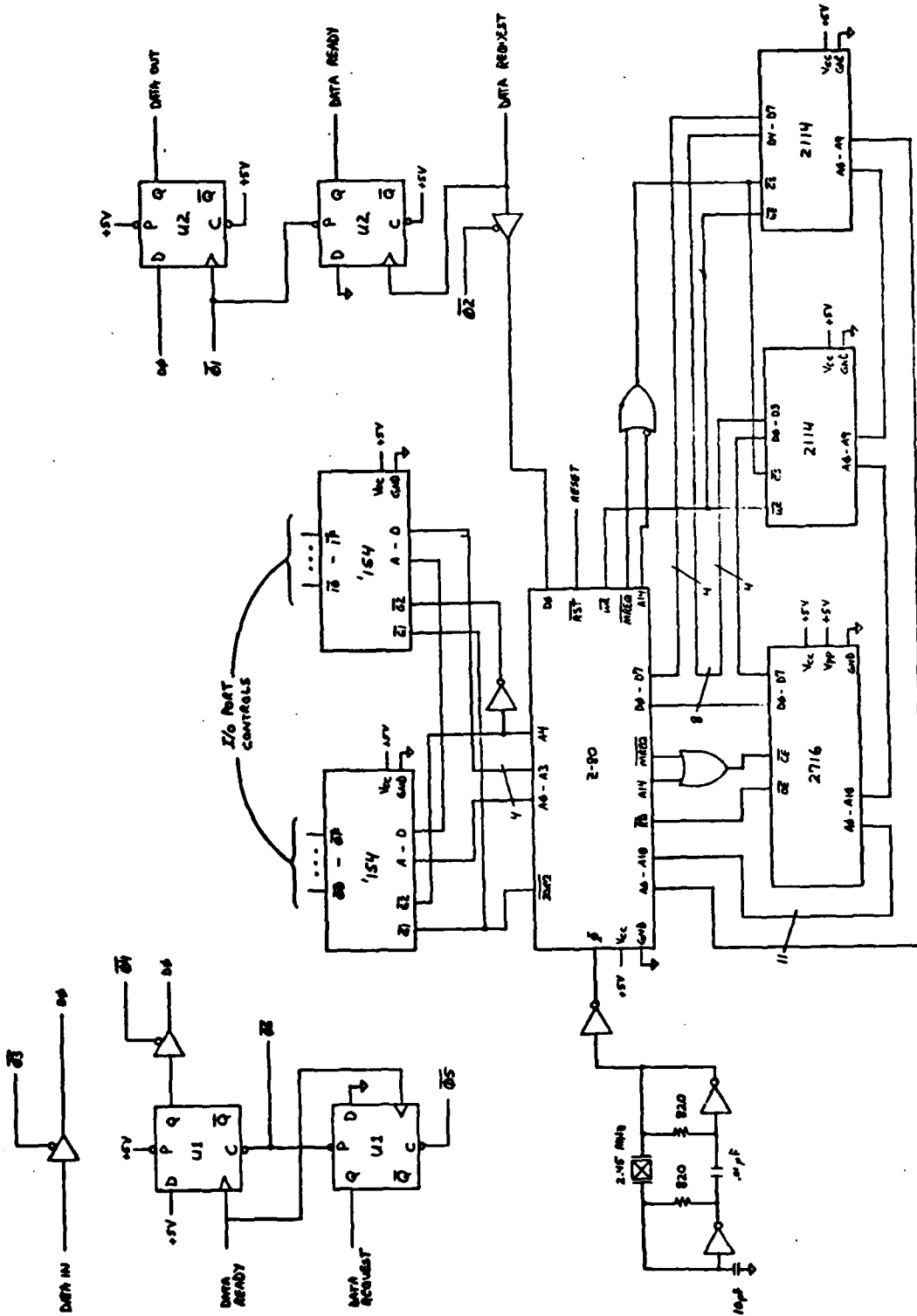


Figure 3. BCH/3olay decoder controller circuit

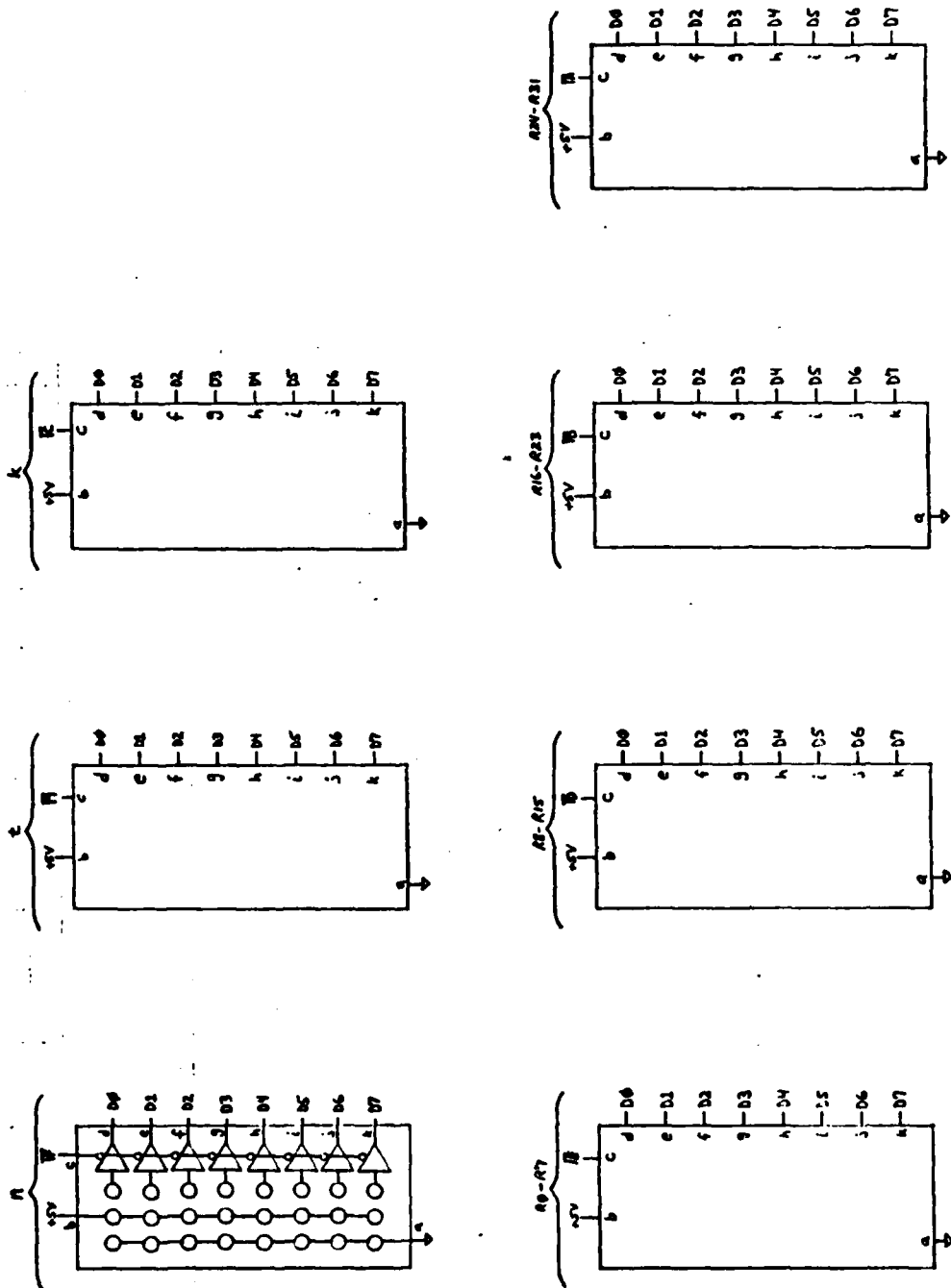


Figure 4. BCH/Golay jumper registers

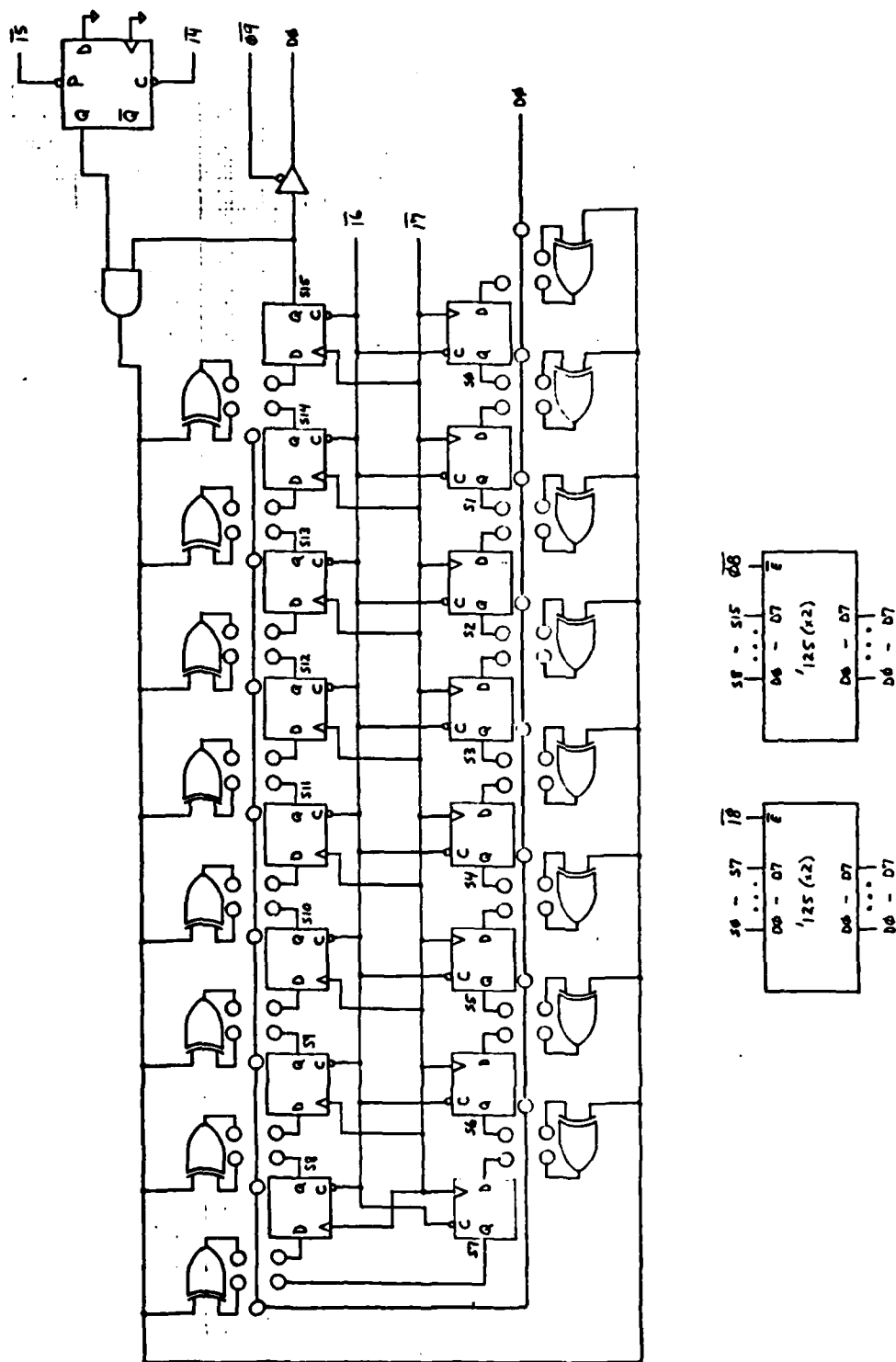


Figure 5. BCH/Golay syndrome register

32 I/O ports (00H to 1FH) is possible.

The jumper registers provide the Z-80 with the values for n , k , and t . They also provide the locations of the k information bits within the n -bit codeword. These values are all set before circuit operations begin. The values for n , k , and t are set in binary. The positions of the information bits are set by jumpering the appropriate locations for positions R0 through R31.

The syndrome register is set up as a divider circuit. The feedback taps to the shift register are set to correspond to the coefficients of the generator polynomial. Thus, as the received codeword is shifted into the syndrome register, it is divided by the generator polynomial. Four '125's (tri-state buffers) are used to gate the contents of the syndrome register onto the data bus. This provides the Z-80 access to the syndrome register.

The following is a step-by-step description of how the decoder circuit works. When a reset signal is received, the Z-80 starts the program from memory location 0000H. The assembly language routine which runs the Z-80 is shown in Appendix B.

Step 1. The controller strobes I/O ports 05* and 16*. This clears the input data request flip-flop U1 and the syndrome register flip-flops.

Step 2. The controller inputs the value for n from I/O port 1F*. This value is preset to the desired input block length (in binary) by jumpers before decoding operations begin.

Step 3. The controller strobes I/O port 06*. This clears the input data ready flip-flop U1 and sets the input data request flip-flop U1.

Step 4. Data bit D0 is input from I/O port 04*, and its value is

checked. This is the input data ready signal. If D0 is zero, step 4 is repeated. If D0 is one, the program proceeds to step 5.

Step 5. Data bit D0 is input from I/O port 03* and stored in memory.

Step 6. The controller checks the total number of data bits stored in memory. If this is less than n , the program goes back to step 3. If this is equal to n , the program proceeds to step 7.

Step 7. The controller strobes I/O port 15*. This turns on the syndrome register feedback gate, causing the syndrome register to function as a divider circuit.

Step 8. The controller shifts each of the received bits stored in memory into the syndrome register one at a time. The bits are shifted into the syndrome register by strobing I/O port 17*.

Step 9. The controller inputs the contents of the syndrome register from I/O ports 18* and 0B*. From these bits the controller calculates the weight of the syndrome register.

Step 10. The controller inputs the value for t from I/O port 19*. If the value for t is greater than or equal to the weight of the syndrome register, the program goes to step 14. If the value for t is less than the weight of the syndrome register, the program proceeds to step 11.

Step 11. The controller checks the total number of times the syndrome register has been rotated after the initial n bits were shifted into the syndrome register. If this number is equal to n , the program goes to step 20. If this number is less than n , the program proceeds to step 12.

Step 12. The controller strobes I/O port 15*. This turns on the

syndrome register feedback gate, causing the syndrome register to function as a divider circuit.

Step 13. The controller outputs a zero bit to I/O port 17*. This causes the contents in the syndrome register to rotate. The controller also rotates the received data bits in memory at the same time. Then the program goes back to step 9.

Step 14. The controller inputs the value for k from I/O port 1C* and the value for n from I/O port 1F*. These values are used to locate the $(n - k)$ bits in memory which correspond to the contents of the syndrome register.

Step 15. The controller strobes I/O port 14*. This turns off the syndrome register feedback gate, causing the syndrome register to function as a shift register.

Step 16. Data bit D0 is input from I/O port 09*. This is the last bit in the syndrome register. This bit is XORed with the corresponding bit in memory.

Step 17. The controller strobes I/O port 17*. This shifts the contents of the syndrome register.

Step 18. The controller checks how many bits have been input from the syndrome register. If this value is less than $(n - k)$, the program goes to step 16. If this value is equal to $(n - k)$, the program goes to step 19.

Step 19. The controller rotates the bits in memory back to their original positions.

Step 20. The controller inputs the locations of the information bits from I/O ports 1E*, 1D*, 1B*, and 1A*. These values are input one

at a time, and the corresponding information bits are located before the next value is input to the controller.

Step 21. Data bit D0 is input from I/O port 02*. This is the output data request line. If D0 is zero, the program repeats step 21. If D0 is one, the program proceeds to step 22.

Step 22. The next information bit is output to I/O port 01*. This sets the output data ready flip-flop U2 and shifts the information bit into the output data flip-flop U2.

Step 23. The controller checks the total number of information bits output to the next circuit. If this value is less than k, the program goes back to step 21. If this value is equal to k, the program goes back to step 1.

Thus, the controller handles the interfacing with other circuits and uses the syndrome register to decode the n-bit codeword into a k-bit information sequence.

Interleaver/Deinterleaver

The interleaver and deinterleaver are implemented by the same circuit and the same assembly language routine. The interleaver takes in 100 bits and retransmits these in groups of ten bits each. The first group is bits 0, 10, 20, . . . , 90; the second group is bits 1, 11, 21, . . . , 91; . . . ; and the tenth group is bits 9, 19, 29, . . . , 99. Thus, the groups are formed by taking every tenth bit from the original 100 bits, starting with each of the first ten received bits. If this process is repeated using the sequence formed by the ten groups of bits as the 100 bits to be interleaved, the resulting sequence will be the original 100-bit sequence. In other words, the interleaver will also

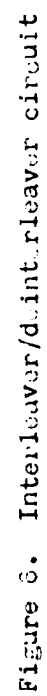
deinterleave its own output. Therefore, the same circuit design is used for the interleaver and the deinterleaver.

The interleaver/deinterleaver is completely implemented by the Z-80 and its associated memory. Figure 6 is a diagram of the interleaver/deinterleaver circuit.

For the EPROM, address lines A0 through A10 are connected to the 2716. The output from ORing A14 and MREQ* is connected to the CE* of the 2716. RD* is connected to the OE*. This set-up provides 2048 memory locations (0000H to 07FFH) for the assembly language routine. Data lines D0 through D7 are connected directly to the 2716. For the RAM, address lines A0 through A9 are connected to both 2114's. The output from ORing A14* and MREQ* is connected to the CS* of the 2114's. This provides 1024 memory locations (4000H to 43FFH) for the Z-80 to use. The WR* signal is connected to the WE* of the 2114's. This controls the read/write operations in the RAM. Data lines D0 through D3 are connected to one 2114, and D4 through D7 are connected to the other 2114. Since each 2114 has only a 4-bit memory word, this provides a full 8-bit memory word for RAM operations. The clock signal for the Z-80 is provided by the crystal-controlled oscillator. The output of the oscillator is a 2.45 MHz square wave.

All of the control functions are implemented through the I/O ports. Address lines A0, A1, and A2 are connected to the '138 (three to eight line decoder), and IORQ* is used to enable the '138. This provides control signals for eight I/O ports (Y0 to Y7*).

The following is a step-by-step description of how the interleaver/deinterleaver works. When the reset signal is received, the



Z-80 starts the program from memory location 0000H. The assembly language routine which runs the Z-80 is shown in Appendix C.

Step 1. The controller strobes I/O port Y3*. This clears the input data ready flip-flop U1 and sets the input data request flip-flop U1.

Step 2. Data bit D0 is input from I/O port Y2*, and its value is checked. This is the input data ready signal. If its value is zero, step 2 is repeated. If its value is one, the program proceeds to step 3.

Step 3. Data bit D0 is input from I/O port Y1* and stored in memory. The data bits are stored sequentially as they are received.

Step 4. The controller checks the total number of bits received. If this value is less than 100, the program goes back to step 1. If this value is equal to 100, the program proceeds to step 5.

Step 5. The controller sets a memory pointer to the location of the first received bit and an offset index to zero.

Step 6. Data bit D0 is input from I/O port Y4*, and its value is checked. This is the output data request signal. If its value is zero, step 6 is repeated. If its value is one, the program proceeds to step 7.

Step 7. The bit in the memory location indicated by the memory pointer plus the offset index is output to I/O port Y5*.

Step 8. The controller checks the number of bits in the present group that have been transmitted. If this value is less than ten, then ten is added to the memory pointer, and the program goes back to step 6. If this value is equal to ten, the program proceeds to step 9.

Step 9. The controller checks the total number of bits transmitted. If this value is less than 100, the memory pointer is set back to the location of the first received bit, the offset index is incremented, and

the program goes back to step 6. If this value is equal to 100, the program goes back to step 1.

Thus, the interleaving and deinterleaving are handled by the Z-80 using RAM to store and rearrange the bit sequences.

Convolutional Encoder

Figure 7 is a diagram of the controller for the convolutional encoder. Figure 8 is a diagram of the convolutional encoder.

The 2716 is the only memory used in this circuit. So the MREQ* signal is connected to the CE* of the 2716. This enables the 2716 during any memory operation. The RD* signal is connected directly to the OE* and is used to gate the contents of the memory onto the data bus. Address lines A0 through A10 are connected to the 2716 to provide 2048 memory locations (0000H to 07FFH) for the assembly language routine. Data lines D0 through D7 are connected directly to the 2716. The clock signal is provided by a crystal oscillator. The output of the oscillator is a 2.45 MHz square wave.

All of the control functions are implemented through the I/O ports. The IORQ* signal is used to enable the '138 (three to eight line decoder) during I/O operations. Address lines A0, A1, and A2 are connected to the three line input on the '138. This provides control signals for eight I/O ports (Y0* to Y7*).

The following is a step-by-step description of how the controller works. When a reset signal is received, the Z-80 starts the program from memory location 0000H. The assembly language routine which runs the Z-80 is shown in Appendix 3.

Step 1. The controller strobes I/O port Y0*. This clears the input

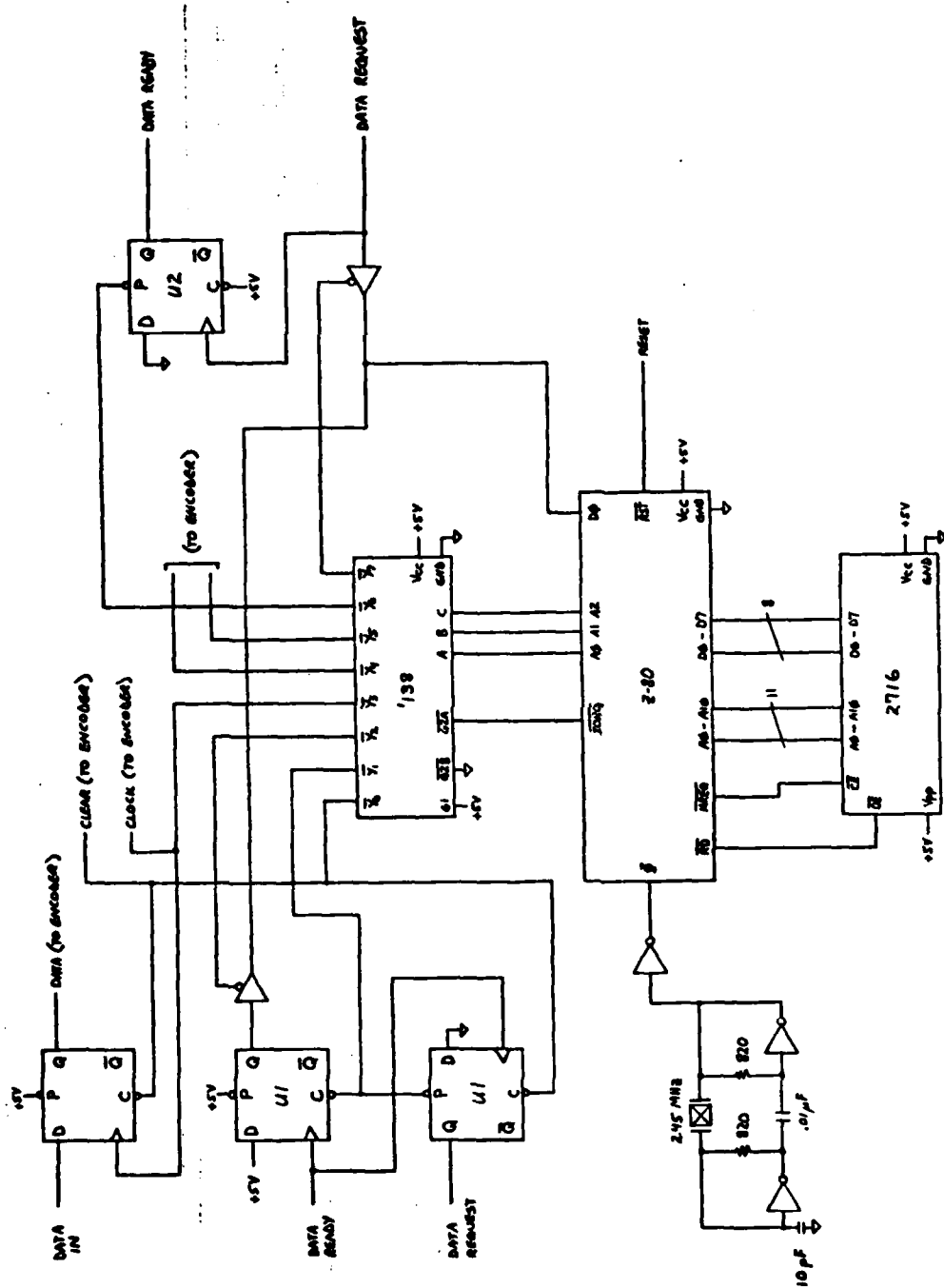


Figure 7. Convolutional encoder controller circuit

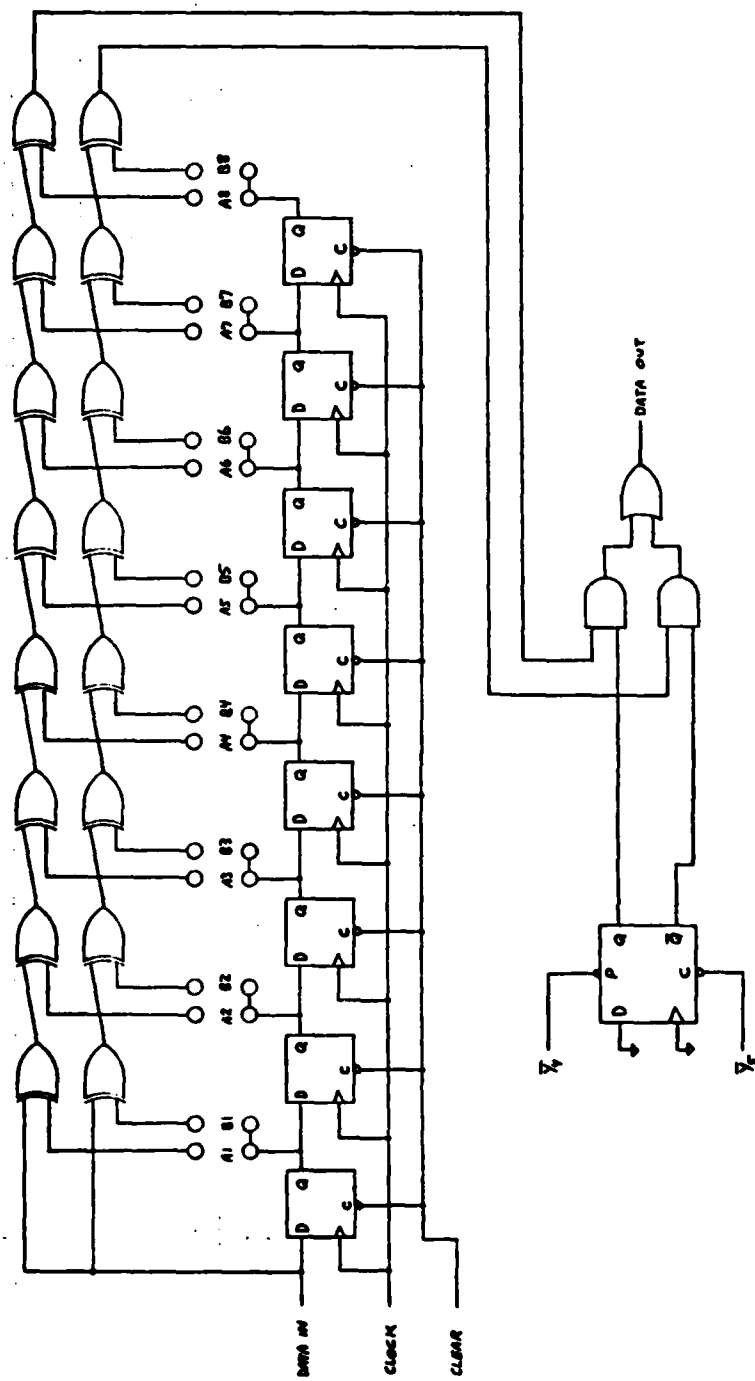


Figure 8. Convolutional encoder circuit

data request flip-flop U1 and the register cells in the encoder.

Step 2. Data bit D0 is input from I/O port Y7*, and its value is checked. This is the output data request signal. If D0 is zero, step 2 is repeated. If D0 is one, the program proceeds to step 3.

Step 3. The controller strobes I/O port Y1*. This clears the input data ready flip-flop U1 and sets the input data request flip-flop U1.

Step 4. Data bit D0 is input from I/O port Y2*, and its value is checked. This is the input data ready signal. If D0 is zero, step 4 is repeated. If D0 is one, the program proceeds to step 5.

Step 5. The controller strobes I/O port Y3*. This clocks the data bit into the encoder circuit.

Step 6. The controller strobes I/O port Y4*. This sets the output gate to pass the first output bit.

Step 7. The controller strobes I/O port Y6*. This sets the output data ready flip-flop U2.

Step 8. Data bit D0 is input from I/O port Y7*, and its value is checked. This is the output data request signal. If D0 is zero, step 8 is repeated. If D0 is one, the program proceeds to step 9.

Step 9. The controller strobes I/O port Y5*. This sets the output gate to pass the second output bit.

Step 10. The controller strobes I/O port Y6*. This sets the output data ready flip-flop U2. Then the program goes back to step 2.

By the previous steps, the controller handles the input and output operations for the encoder. The encoder itself is a shift register and two modulo two adder circuits. The inputs to the adder circuits are the jumpered outputs of each cell of the shift register. The jumpers for

each adder circuit are set to correspond to the coefficients of the generator sequences for the desired code. This circuit will implement any $(2, 1, m)$ convolutional code for $m < 9$.

IV. Operations

Data Transfer

The circuits have been designed to operate in an asynchronous manner. Each circuit has its own microprocessor controller and clock generator. Thus, the internal operations (as described in the previous chapter) are independent of the other circuits. Data transfer between the circuits is handled through three interface signals: data, data request, and data ready.

Data transfer between circuits is handled in the following manner. Consider two circuits A and B, and assume data is to be transferred from circuit A to circuit B. First, circuit B sets the data request line to +5 V. This clears the data ready line and notifies circuit A that circuit B is ready to accept the next data bit. The data ready line is cleared so that circuit B will not see the data ready signal until circuit A has acknowledged this specific data request. After circuit A has received the data request and when it is ready to transfer the next data bit, it sets the data ready line to +5 V. This clears the data request line and notifies circuit B that a valid data bit is ready to be transferred. Circuit B then latches the data bit from the data line, and the cycle will start again when circuit B is ready to request another data bit.

The only other signal which is connected between the circuits is the reset signal. This signal is generated by the computer controlling the input and output of data to and from the concatenated coding set-up. When this signal is reset to 0 V (idle state is +5 V), each circuit in the system is cleared. When a circuit is cleared, all registers are set

to zero, all data request lines are set to 0 V, and the controller initializes the circuit.

General Configuration

Because each circuit functions independently of the others, the encoding and decoding circuits may be concatenated in any order. Of course the decoders must be connected in an order which will properly decode the output from the encoders. In other words, the order in which the decoders are connected must be exactly opposite the order in which the encoders are connected. After each circuit has been configured for the desired codes (as described in Chapter III), they can be concatenated by connecting the interface and reset signals of the adjacent circuits. Thus, once the circuits are connected to form the concatenated encoder, the flow of data into and out of the system is controlled by the computer.

The basic set up will use the computer as the source of data bits. These bits will be transferred from the computer to the concatenated encoders. The output from the encoders will be transferred to the computer simulated channel, and the output from this channel will be transferred to the concatenated decoders. The decoded output will be transferred back to the computer for processing.

V. Conclusions and Recommendations

Conclusions

Of the codes considered, the BCH encoder and decoder, the Golay encoder and decoder, the convolutional encoder, the interleaver, and the deinterleaver were all implemented in hardware. The Reed-Solomon encoder and decoder and the Viterbi decoder were not implemented.

Recommendations

The Eclipse computer should be interfaced to the encoding and decoding circuits via serial I/O ports. The following software should be implemented on the Eclipse computer:

1. Reed-Solomon encoder and decoder
2. Viterbi decoder
3. Random bit generator to generate the bit sequences to be transmitted through the system
4. Error channel simulator
5. Error performance calculator

Additionally, as stated in Chapter II, a Kasami or systematic search decoder may be implemented on the Eclipse computer.

Appendix A

BCH/Golay Encoder Assembly Language Routine

START:	OUT(00H),A	clear data request and encoder, enable data gate
	IN A,(06H)	load n (output block length)
	LD B,A	B = output block length
	IN A,(07H)	load k (input block length)
	LD C,A	C = input block length
LOOP:	IN A,(05H)	check data request
	AND 01H	mask bit
	JP Z,LOOP	no request, check again
	OUT (01H),A	send request, clear ready line
NEXT:	IN A,(02H)	check data ready
	AND 01H	mask bit
	JP Z,NEXT	not ready yet, check again
	OUT (03H),A	clock in data, send data ready
	DEC B	B = B - 1
	DEC C	C = C - 1
	JP NZ,LOOP	not k bits yet, get next bit
	OUT (04H),A	set data gate to zero
CHECK:	IN A,(05H)	check data request
	AND 01H	mask bit
	JP Z,CHECK	no request, check again
	OUT (03H),A	clock zero into encoder, send data ready
	DEC B	B = B - 1
	JP NZ,CHECK	not n bits yet, send next bit
	JP START	done, start again

Appendix B

BCH/Golay Decoder Assembly Language Routine

STARTUP:	OUT (05H),A OUT (16H),A LD SP,4200H IN A,(1FH) LD C,A LD B,A SUB 01H LD E,A LD HL,4100H	clear data request line clear syndrome register initialize SP to 4200H load n into A set index C to n set index B to n $A = n - 1$ load $n - 1$ into E initialize memory pointer to 4100H
LOAD:	CALL LOADBIT DEC B JP NZ,LOAD CALL CALC	load in a data bit decrement index not n bits, get next bit n bits, calculate syndrome
WEIGHT:	LD D,00H IN A,(18H) CALL CHECK IN A,(0BH) CALL CHECK IN A,(19H) SUB D JP M,GRTR JP ADDER DEC C JP M,SETUP	clear D load S0-S7 check weight. load S8-S15 check weight. load threshold value t $A = t - (\text{syndrome weight})$ check sign of t - (syndrome weight) t is greater or equal to syndrome weight decrement index counter check for n total rotates
GRTR:	CALL ROTATER CALL ROTATES JP WEIGHT	rotate received bits rotate syndrome register check syndrome register weight again
ADDER:	IN A,(1CH) LD D,A IN A,(1FH) SUB D SUB 01H LD E,A LD D,00H SBC HL,DE OUT (14H),A	load A with k load D with k load A with n $A = n - k$ $A = n - k - 1$ $E = n - k - 1$ clear D $HL = HL - (n - k - 1)$ set syndrome register to shift
LOOP:	IN A,(09H) AND 01H XOR (HL) LD (HL),A INC HL DEC E JP M,READY OUT (17H),A JP LOOP	input syndrome bit mask off bit $A = \text{received bit XOR syndrome bit}$ store result in memory point to next received bit check for last bit done, set up for output shift syndrome register get next bit

READY:	DEC HL	set memory pointer
NEXT:	DEC C	check total number of rotates
	JP M, SETUP	n rotates made, find info bits
	CALL ROTATER	rotate received bits
	JP NEXT	check for n rotates
SETUP:	LD DE, 4005H	set memory location to 4005H
	IN A, (1FH)	load A with n
	LD C, A	load C with n
	IN A, (1EH)	load A with info set 1
	CALL FIND	find info bits in received bits
	IN A, (1DH)	load A with info set 2
	CALL FIND	find info bits in received bits
	IN A, (1BH)	load A with info set 3
	CALL FIND	find info bits in received bits
	IN A, (1AH)	load A with info set 4
	CALL FIND	find info bits in received bits
	JP STARTUP	restart routine
LOADBIT:	OUT (06H), A	send data request
LDBA:	IN A, (04H)	check data ready
	AND 01H	mask bit
	JP Z, LDBA	not ready, check again
	IN A, (03H)	input data
	AND 01H	mask bit
	LD (HL), A	store bit in memory
	DEC HL	point to next position in memory
	RET	return to main program
CALC:	OUT (15H), A	set syndrome register to rotate
CLCA:	INC L	set memory pointer
	LD A, L	load L into A
	SUB 01H	$A = L - 1$
	JP Z, CLCB	check for last received bit
	LD A, (HL)	load received bit
	OUT (17H), A	shift received bit into syndrome register
	JP CLCA	get next bit
CLCB:	DEC L	reset memory pointer
	RET	return to main program
CHECK:	LD B, 08H	set index to 08H
CHKA:	RRA	check next bit
	JP Z, CHKB	if zero, skip add
	INC D	increment weight value
CHKB:	DEC B	decrement index
	JP NZ, CHKA	if not zero, check next bit
	RET	return to main program
ROTATER:	LD D, 00H	clear D
	SBC HL, DE	$HL = HL - (n - 1)$: point to bottom of memory

	LD A,(HL)	load bit into A
	ADC HL,DE	HL = HL + (n - 1): point to top of memory
	INC HL	point to top of memory plus one
	LD (HL),A	store bit in memory
	RET	return to main program
ROTATES:	OUT (15H),A	set syndrome register to rotate
	AND 00H	clear A
	OUT (17H),A	rotate syndrome register
	RET	return to main program
FIND:	LD B,08H	set index to 08H
FNDA:	DEC B	decrement index
	JP M,FNDC	check for last bit
	DEC C	decrement counter
	JP M,FNDD	check for n bits
	RRA	check next bit position
	JP NC,FNDB	if zero, check next bit
	LD 4000H,A	store A in memory (4000H): store info
		set
	LD A,(HL)	load A with received bit
	LD (DE),A	store bit in memory
	INC DE	increment memory pointer
	LD A,4000H	load A with info set
FNDB:	DEC HL	point to next received bit
	JP FNDA	check for next info bit
FNDC:	RET	return to main program
FNDD:	IN A,(1CH)	load A with k
	LD B,A	load B with k
	LD DE,4005H	set memory pointer to info bits starting location
FNDE:	CALL SENDBIT	output info bit
	DEC B	decrement index
	JP NZ,FNDE	if not zero, send next bit
	RET	return to main program
SENBIT:	IN A,(02H)	check data request
	AND 01H	mask bit
	JP Z,SENBIT	no request, check again
	LD A,(DE)	load A with info bit
	INC DE	point to next bit
	OUT (01H),A	output info bit
	RET	return to main program

Appendix C

Interleaver/Deinterleaver Assembly Language Routine

STARTUP:	LD HL,4100H LD SP,4200H LD D,64H OUT (03H),A	set memory pointer set stack pointer set index to 100 request data
CHECK :	IN (02H),A AND 01H JP Z,CHECK CALL LOADBIT DEC D JP Z,OUTPUT JP CHECK	check data ready bit mask bit not ready, check again input data bit decrement index 100 bits in, go to output not 100 bits
OUTPUT:	LD HL,4100H LD C,00H LD D,0AH LD B,0AH	set memory pointer clear low pointer set offset set group index
NEXT:	LD E,0AH	set bit index
LOOP:	IN A,(04H) AND 01H JP Z,LOOP LD A,(HL) OUT (05H),A LD A,L ADD D LD L,A DEC E JP Z,ONE JP LOOP	input data request mask bit no request, check again get bit at memory pointer output bit get low order memory add offset to low order memory set new value in memory pointer decrement bit index if group done, check total bits group not done, send next bit
ONE:	DEC B JP Z,STARTUP INC C LD L,C JP NEXT	decrement group index 10 groups done, start again increment low pointer set low order memory send next group
LOADBIT:	IN A,(01H) LD (HL),A INC HL RET	get data bit store bit in memory increment memory pointer return to main program

Appendix D

Convolutional Encoder Assembly Language Routine

CHECK1:	OUT (00H),A IN A,(07H) AND 01H JP Z,CHECK1 OUT (01H),A	clear input data request get output data request mask bit no request, check again request data
CHECK2:	IN A,(02H) AND 01H JP Z,CHECK2 OUT (03H),A OUT (04H),A OUT (05H),A	input data ready mask bit not ready, check again clock in data set output to code bit 1 send data ready
CHECK3:	IN A,(07H) AND 01H JP Z,CHECK3 OUT (05H),A OUT (06H),A JP CHECK1	get output data request mask bit no request, check again set output to code bit 2 send data ready set up for next input bit

Bibliography

Cited Reference

1. Costello, D. J. and Lin, S. Error Control Coding: Fundamentals and Applications. Englewood Cliffs, NJ: Prentice Hall, Inc., 1983.

Supplemental References

Blahut, R. E. Theory and Practice of Error Control Codes. United States of America: Addison-Wesley Pub. Co., 1983.

Gallager, R. G. Information Theory and Reliable Communication. United States of America: John Wiley and Sons, Inc., 1968.

Pless, V. Introduction to the Theory of Error Correcting Codes. New York, NY: Wiley Interscience, 1982.

VITA

Captain Peter W. de Graaf received his commission in May 1981. His first assignment was as a student in the Communications-Electronics Engineer Course at Keesler AFB. In January 1982, he was reassigned to the 1815 Test and Evaluation Squadron at Wright-Patterson AFB. While with the 1815th, he served as an evaluation team engineer and worked in the AFCC Systems Evaluation School. After his assignment with the 1815th, he entered the School of Engineering in June 1984.

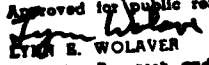
Permanent address: RD1 Box 272

Dingman's Ferry, PA 18328

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1d. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GE/ENG/85D-12			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION School of Engineering		6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION		
6c. ADDRESS (City, State and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, OH 45433			7b. ADDRESS (City, State and ZIP Code)		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Foreign Technology Division		8b. OFFICE SYMBOL (If applicable) FTD/TQCS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c. ADDRESS (City, State and ZIP Code) FTD Wright-Patterson AFB, OH 45433			10. SOURCE OF FUNDING NOS.		
11. TITLE (Include Security Classification) See Box 19			PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.
			WORK UNIT NO.		
12. PERSONAL AUTHOR(S) Peter W. de Graaf, Capt, USAF					
13a. TYPE OF REPORT MS Thesis		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Yr., Mo., Day) 1985 December	
15. PAGE COUNT 48					
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB. GR.	Concatenated Error Coding, Block Coding Convolutional Coding, Error Detection and Correction		
C9	C4				
19. ABSTRACT (Continue on reverse if necessary and identify by block number)					
Title: HARDWARE IMPLEMENTATION OF A CONCATENATED ENCODER/DECODER					
Thesis Advisor: Glenn E. Prescott, Capt, USAF Professor of Electrical Engineering					
<p>Approved for public release: LAW AFR 100-14.  LYNN E. WOLAVER 16 JAN 86 Dean for Research and Professional Development Air Force Institute of Technology (AFIT) Wright-Patterson AFB OH 45433</p>					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS <input type="checkbox"/>			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL Glenn E. Prescott, Capt, USAF			22b. TELEPHONE NUMBER (Include Area Code) (513) 255-3576		22c. OFFICE SYMBOL AFIT/ENG

DD FORM 1473, 83 APR

EDITION OF 1 JAN 73 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

This study describes the hardware implementation of a concatenated error correcting encoder/decoder. Individual burst and random error correcting codes were implemented using standard TTL integrated circuits and Z-80 microprocessors. The circuits handle input and output operations with a three line handshake. Thus, data transfer between circuits is asynchronous, and the coders can be concatenated in any order.

Reed-Solomon, BCH, Golay, interleaving, and convolutional codes were considered. Of these codes, The BCH encoder/decoder, the Golay encoder/decoder, the interleaver/deinterleaver, and the convolutional encoder were all implemented in hardware. The Reed-Solomon encoder/decoder and the convolutional decoder will be implemented in a follow-on study in software.

This study is the first part of a group of studies which will ultimately determine the actual error detection and correction performance of various concatenated coding schemes.

END
FILMED

5-86

DTIC